UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Java I – Pós-Graduação Tecnologia Java
Atividade 09 – Revisão de Conteúdo
**Aluno: Camilo Cunha de Azevedo**

---

1. Orientação à Objetos é um Paradigma que antecede linguagens como Java e C++, sendo criado inicialmente por Alan Kay junto com a Linguagem Smalltalk que era puramente orientada a objetos. Java não satisfaz todos os requisitos de uma linguagem Orientada a Objetos propostos por Kay e por tanto não é totalmente aderente a todas as práticas de Orientação a Objetos.

   Um dos principais pontos nisso é que para uma linguagem ser totalmente Orientada a Objetos, tudo deve ser um Objeto, e nisso, Java não atende pois trabalha internamente com os tipos primitivos da JVM. Além de trazer conceitos que não fazem parte do paradigma como encapsulamento (em Smalltalk que é um exemplo de linguagem puramente Orientada a Objetos tudo é público a nível de sistema no que é conhecido como *Live Environment).*

   Porém, apesar disso, ela possui um Design que favorece o uso de Classes e Objetos e permite o uso de principais pontos nisso é que para uma linguagem ser totalmente Orientada a Objetos para o reaproveitamento de código utilizando as melhores práticas e padrões do mercado. Isso se deve graças a ela ter implementado a maior parte do conceito de Orientação a Objetos de C++, substituindo partes como Herança Múltipla por conceitos que foram considerados mais interessantes. E nesse sentido, é plenamente aderente a Orientação a Objetos (no mesmo sentido que diríamos que C++ é plenamente Orientada a Objetos).

2. Abstração é a técnica de transpor as características e comportamentos relevantes de um Objeto para sua representação em código.

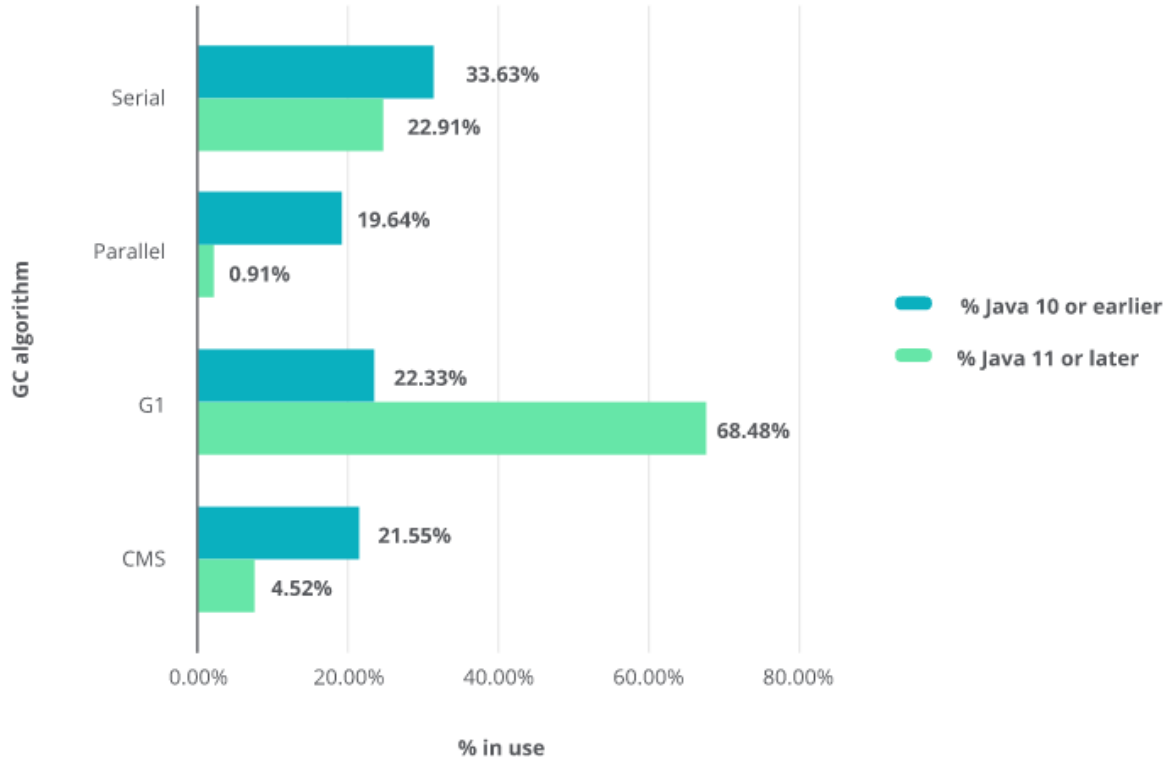
3. Objetos, Classes, Atributos e Métodos.

4. Tipos Primitivos são valores primitivos da Máquina Virtual Java que representam valores em si (que não são objetos). Sendo eles cinco valores numéricos byte, short, int, long e char, o valor de boolean que faz o encoding de valores true e false e os valores de Ponto Flutuante float e double. Cada um desses tipos tem o seu limite e especificação definido na especificação da JVM.

5. Um tipo abstrato é um tipo que não possui uma implementação concreta, ele serve como um “molde” ou como um “contrato” para os tipos concretos que o implementam.

6. Garbage Collector é um modelo de algoritmo que é responsável por limpar a memória que é alocada por programas de uma determinada linguagem de programação. Em Java, a JRE traz vários algoritmos de Garbage Collection, sendo que o algoritmo padrão é escolhido pela especificação do fabricante da JRE, hoje o mais utilizado nas versões recentes de Java segundo pesquisa da New Relic que postei  no Discord da Especialização é o Algoritmo G1.

   O uso de Garbage Collector serve para dinamizar a alocação de memória, deixando o programador livre dessa responsabilidade podendo focar melhor na qualidade de código e na regra de negócio que está implementando.




(Popularidade do uso dos Algoritmos de Garbage Collection segundo pesquisa da New Relic nas versões 10 e 11 de Java.)

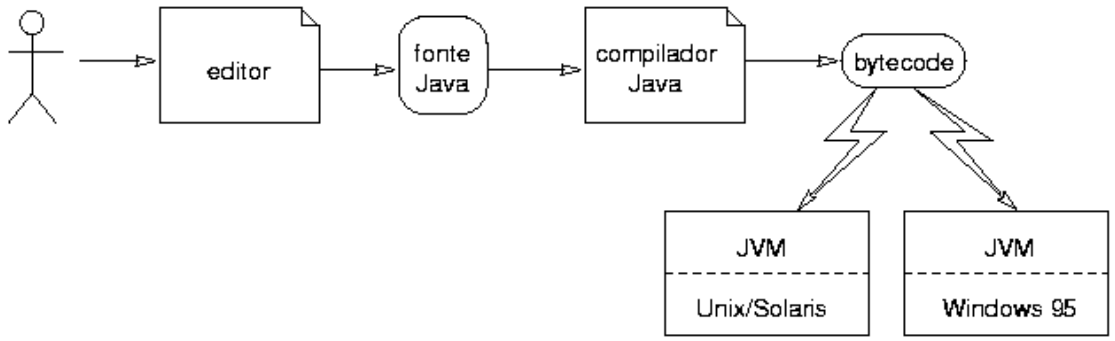
7. **a.** Em qualquer Sistema Operacional (isso inclui o Windows) que tenha a JDK instalada e configurada no PATH/Chaves de Registro, se usa a ferramenta "**javac**" (Java Compiler) para compilar o arquivo para Bytecode da JVM passando o caminho do arquivo fonte como parâmetro para esse comando.

   b. Para executar uma aplicação em qualquer Sistema Operacional (isso inclui o Windows) que tenha a JDK instalada e configurada no PATH/Chaves de Registro, usamos a ferramenta "**java**" passando como parâmetro o caminho do arquivo que contêm os Bytecodes que a JVM deve executar.

8. Bytecode é uma representação de instruções da Máquina Virtual Java. Cada Bytecode representa uma instrução de operação intermediária na qual a JVM traduz de forma otimizada para as instruções nativas da máquina. Quando um código-fonte é compilado usado a ferramenta "javac" o código é compilado para

bytecode que podem ser visualizados usando a ferramenta “javap” que vem junto a instalação da JDK.

**9.** A tradução de bytecodes permite a portabilidade de código Java, o famoso lema “escreva uma vez e rode em qualquer lugar” uma vez em que o código em bytecode pode ser executado em qualquer máquina que tenha uma JRE instalada independente do conjunto de instruções do Sistema Operacional/Processador.




(Esquema de compilação/execução de bytecode na JVM)

a. O arquivo .java é o arquivo fonte, que contém o código Java, esse pode ser compilado para bytecode em qualquer computador que tenha uma JDK instalada.

b. O arquivo .class é o arquivo que resulta da compilação do código-fonte que contém os bytecodes da JVM, que ela irá utilizar para traduzir (interpretar) para instruções da máquina local.

c. O Sistema Operacional é o software responsável por gerenciar e administrar todos recursos presentes em uma máquina e/ou sistema. É da responsabilidade do Sistema Operacional executar os programas que são executados na máquina e diferentes Sistemas Operacionais possuem diferentes modos de trabalhar isso (ELF, PE…).

d. A JVM (Java Virtual Machine) é uma Máquina Virtual que é utilizado para executar programas compilados para seu Bytecode independente de plataforma. Os programas serem executados em

uma Máquina Virtual permite além da portabilidade de código graças a tradução de bytecode também uma especificação neutra para a execução dos programas, sendo padronizado por exemplo o acesso a arquivos, o número de bytes dos tipos e o encoding de Strings independente do Sistema Operacional em que o programa será executado.

**10.** Desde o ínicio a linguagem/plataforma Java foi projetada com a finalidade de tornar alguns ataques impossíveis tais quais como a sobrescrita de memória em execução (muito comum em vírus de computador), corrupção de memória fora de seu espaço de memória ou leitura/sobrescrita de arquivos sem permissões.

Entre as principais medidas que foram tomadas nesse sentido estão a utilização de referências ao invés de ponteiros impedindo o acesso direto a memória e o controle direto da JVM a nível de sistema operacional (níveis de segurança de Hardware), assim como o uso de APIs que foram projetadas levando em conta um nível excelente de segurança das aplicações (níveis de segurança de Software).

**11.** A modularidade é a capacidade de parametrizarmos o código através de relações entre classes. Isso nos permite isolar, compartilhar e modelar dados e comportamentos entre diferentes classes e seus objetos de instância.

a. Isso ajuda na manutenção do código permitindo melhor manutenção, celeridade e simplicidade de código graças ao correto projeto e execução seguindo os princípios de modularidade.

Coesão é restringir os módulos para uma responsabilidade única.

Acoplamento significa desenvolver a relação entre os módulos de forma que funcionem de forma a não depender de abstrações externas.

**12.** a. this representa o objeto em si.
b. super representa o objeto pai/estendido.

**13.** **OBS**: *Vou criar os exemplos usando Carbon para melhor visualização*.

```
public abstract class Carro {
    String cor;

    public String getCor() {
        return this.getCor().toUpperCase();
    }
}
```

```
public class Gol extends Carro {
    int ano;

    public Gol(int ano, String cor) {
        // chama cor de super (super = Carro)
        super.cor = cor;

        // chama ano de this (this = Gol)
        this.ano = ano;
    }

    @Override
    public String getCor() {
        return super.getCor().toLowerCase();
    }
}
```

**14.**

a. **Encapsulamento** é um conceito presentes em linguagens como C++ e Java que implementam controle de acesso a nível de módulo (pacote). Serve para garantir que determinadas partes do código só possam ser acessadas

com a devida autorização de acesso garantindo maior segurança de codificação e execução do código.

   i. O encapsulamento pode ter 3 níveis de acesso:
      1. **Público**: Que dá o acesso e visibilidade a qualquer outra parte do código ou chamada.
      2. **Protegido**: O acesso e visibilidade é dado somente a membros que estejam dentro da própria classe/módulo.
      3. **Privado**: Que permite o acesso somente dentro da própria classe. A visibilidade é restrita à própria classe.
   ii. O encapsulamento ajuda a aumentar a segurança do código restringindo o acesso permitindo esconder partes sensíveis e padronizando a chamada de atributos e métodos.

b. Herança é a capacidade de uma Classe extender outra, ou seja se tornando uma parte extensão que contém os mesmos atributos e métodos da classe extendida. Em Java e C++ isso entra dentro do conceito de Subtyping.
   i. Especialização é o caso em que uma classe estende outra herdando todos os membros da superclasse e adicionando seus próprios.

      Generalização é a classe “acima” ou superclasse que é o conceito geral das classes que são herdadas dela.
   ii. A herança ajuda a padronizar o código uma vez em que parametriza um comportamento padrão compartilhado pelas classes que a extendem (ou herdam), além de permitir reaproveitar o código e ter maior controle sobre o sistema o que eventualmente leva a maiores níveis de segurança.
   iii. Quando uma classe herda o comportamento de outra ela recebe tudo o que a superclasse possui, assim podendo levar a um código mais terso e a capacidade de reaproveitamento de código.

c. Polimorfismo é a capacidade de uma subclasse poder ser usada em qualquer aplicação em que a superclasse pode

ser usada. Em Java e C++, quando aliamos o Polimorfismo ao Subtyping temos o Polymorphic Subtyping que é definido na Ciência da Computação como: “Se o elemento S é do subtipo T, pode-se seguramente usar S onde T é esperado.”

i. Sobrecarga é quando redefinimos um método com o mesmo nome, porém com uma assinatura diferente. Isso permite que coexistam múltiplos comportamentos dependendo dos parâmetros usados na chamada.
ii. Sobrescrita é quando o comportamento de um método da superclasse é alterado em uma classe filha, seja para a especialização ou para um novo comportamento.
iii. Coerção é quando forçamos a conversão de um tipo a outro em que seja possível o casting (de um tipo com tamanho menor ou igual para um tipo de tamanho maior ou igual).

**15.**

```java
public class Pessoa {
    // Encapsulamento
    private String nome;

    // Construtor
    public Pessoa() {
        this.nome = "";
    }

    // Sobrecarga do Construtor
    public Pessoa(String nome, long cpf) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public void imprimeDados() {
      System.out.println(nome);
    }
}
```

```java
public class PessoaJuridica extends Pessoa {
    private String cnpj;

    public PessoaJuridica() {
        this.nome = "";
    }

    // Sobrecarga do Construtor
    public PessoaJuridica(String nome, String cnpj) {
        super(nome);
        this.cnpj = cnpj;
    }

    public String getCnpj()) {
        return cnpj;
    }

    public void setCnpj(String cnpj) {
        Boolean valido = valida(cnpj);

        if(valido) {
            this.nome = nome;
        } else {
          System.out.println("CNPJ Inválido");
        }
    }

    // Sobreescrita
    public void imprimeDados() {
      System.out.println("Nome: " + nome + " CNPJ " + cnpj);
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
      Pessoa pessoa;
      PessoaJuridica utfpr = new PessoaJuridica("UTPFR", "001.234.123/0001-90");

      // Coerção
      pessoa = (Pessoa) utfpr;
      utfpr.imprimeDados();
    }
}
```

**16.** O conceito de “trocas de mensagens” foi proposto por Alan Kay quando criou o conceito de Orientação a Objetos em que a mensagem seria um valor passado de uma Classe a outra através de Métodos. Esse é um conceito com sentidos diferentes em

linguagens puramente Orientadas a Objeto como Smalltalk ou Self do que em linguagens que tem uma mistura de imperativo e OO como Java e C++.

Em Smalltalk (e Self, ou qualquer outra linguagem puramente OOP) a mensagem passada de um Objeto a outro pode ser qualquer coisa e o objeto que está recebendo o valor é que decide o que fazer (como operar a partir disso seja descartando o valor ou realizando uma computação) em tempo de execução.

Em Java ou C++ a passagem de mensagens é a invocação de métodos passando valores por argumentos, porém, essas linguagens apenas suportam a invocação de métodos conhecidos de antemão pelo compilador passando argumentos esperados que são analisados em tempo de compilação e somente executados em tempo de execução.

**17.** O método construtor é o método utilizado para “construir” (ou seja, chamado na criação) às instâncias, ele é executado assim que uma nova instância é construída e geralmente é utilizado para popular os dados necessários. Quando não declarado o compilador irá instanciar um construtor vazio de forma implícita durante a fase de compilação.

```java
public class Gato {
    String nome;
    String especie;
    int idade;

    // construtor vazio chamado quando = new Gato();
    public Gato() {
    }

    // construtor que popula o nome, chamado quando = new Gato("Galileu");
    public Gato(String nome) {
        this.nome = nome;
    }

    // construtor que popula todos os atributos da Classe.
    // chamado quando = new Gato("Zelda", "Vira Lata", 6);
    public Gato(String nome, String especie, int idade) {
        this.nome = nome;
        this.especie = especie;
        this.idade = idade;
    }
}
```

**18.** a. Classe Abstrata é uma classe que não possui implementação concreta, ela não pode ser instanciada em um Objeto e tem como principal função representar conceitos abstratos como ideias que serão estendidos por outras classes.

b. Métodos Abstratos são métodos que servem como um “contrato” para que a classe filha de onde são descritos implementam o comportamento específico de determinado método.

c. Classe Final é uma classe que é o “fim” de uma árvore de relações de herança, ou seja, não pode ser herdada por nenhuma outra Classe. Isso serve para garantir que o sistema não terá extensões do comportamento da mesma.

d. Atributos Finais, também conhecidos como atributos constantes, são atributos que não podem ter seu valor alterado.

e. Métodos Finais são métodos que não podem ser sobrescritos, isso é muito importante quando é necessário ter a certeza que o comportamento de determinado método não será alterado pelas classes que o herdam.

**19.** Interfaces é um conceito presente em Java como uma forma de resolver problemas de Design presentes em linguagens Orientadas a Objetos anteriores como C++ e SIMULA que possuiam herança múltipla. Interfaces representam um comportamento que é da classe e que a classe é esse comportamento (funcionando como uma “herança” que pode ser usada com Polimorfismo), porém, ela apenas define comportamentos que a Classe que a implementa deve ter, sem sua especificação concreta para que obrigue todas as classes que compartilham algum comportamento a implementá-lo.